Checking Temporal Relations between Multiple Objects

Ciera Jaspan[*] **Jonathan Aldrich**[*]

June 2008[†]
CMU-ISR-08-119

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[*]School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA
[†]Originally written December 2007

This work was supported in part by NSF grant CCF-0546550, DARPA contract HR00110710019, the Department of Defense, Army Research Office grant number DAAD19-02-1-0389 entitled "Perpetually Available and Secure Information Systems", and the Software Industry Center at CMU and its sponsors, especially the Alfred P. Sloan Foundation.

Keywords: software frameworks, object constraints, relationships, static analysis, shape analysis

Abstract

Software frameworks contain constraints with unusual properties. The constraints frequently span multiple objects and classes, and they change according to the current context. Additionally, they may not be enforced at the same point where they were specified or broken, thus causing unexpected runtime errors. This paper describes a lightweight specification system to describe multi-object temporal constraints. It also provides a detailed description of a static analysis to check that the constraints are not broken. The implementation of the analysis is used to check example plugins from the ASP.NET and Eclipse frameworks.

1 Introduction

Object-oriented frameworks have brought many benefits to software development, including reusable codebases, extensible systems, and encapsulation of quality attributes. However, frameworks are used at a high cost; they are complex and difficult to learn [1]. This is partially due to the complexity of the constraints they place on the *plugins* that utilize them. These constraints typically concern multiple objects that interact together and change based upon the state of the plugin or framework.

The complexity of these constraints makes them both difficult to understand and difficult to document. Since multiple objects are involved, it is not clear where the documentation should reside so that plugin developers will find it. The documentation might be associated with the method which enforces the constraint, but it could be broken by a different method, possibly in a different class. It is also possible that the constraint only matters within certain contexts, in which case, the documentation must be quite lengthy to describe when an operation is and is not allowed.

When a constraint does break, it can be difficult to find the location of the error. A broken constraint frequently results in unexpected exceptions or unusual runtime behavior, and the erroneous behavior might occur long after the constraint was initially broken. Since the constraint may span multiple objects, the object that enforces the constraint may not be the same object that broke the constraint. For these reasons, even experienced plugin developers have difficulty keeping track of all the constraints with which they must comply.

In previous work [2], we proposed a preliminary specification and sketched a hypothetical analysis to discover mismatches between the plugin code and the declared constraints of the framework. The previous work primarily discussed the requirements for such a system and explored a prototype specification with an example. Previous work also introduced the following principles that guide our system:

1. *No up-front effort for plugin developers.* The plugin developers should not have to make any additional effort to use the framework. In particular, they do not add any specifications to their plugin code.

2. *Minimal effort for the framework developer.* The framework developer will specify, in a concise notation, how to use the framework and what constraints exist.

3. *Localized errors.* Frameworks require developers to use many objects from different places in the framework. Currently, errors from a framework constraint frequently do not correspond to the location where the constraint was broken; they instead direct the user to the location where the constraint was enforced. The warnings from the analysis should direct the plugin developer to the line where the error occurred, not the line where it is enforced.

4. *Modularity.* Framework constraints work across many objects, and some objects are governed by many constraints. It should be possible to specify each constraint separately. Framework developers should not be forced to specify the entire framework, or even an entire class. The analysis should be modular and produce usable results based on whatever specifications exist.

In this paper, we provide four contributions:

1. We introduce the concept of developer-defined relations across objects and show that it captures the programming model used to interact with frameworks. We provide two examples from different frameworks to show that relations describe the intuition behind common interaction patterns. (Section 2)

2. We have created a way to describe framework constraints in a concise manner while increasing the expressiveness from the previous proposal. The constraints contain both a syntactic and semantic part, so that they can express framework constraints that are temporal and work across multiple objects. (Section 3)

3. We provide the intuition and technical details for a local, static analysis that detects broken constraints in plugins. We also provide a discussion about when false positive and false negatives can occur. (Section 4)

4. We implemented the static analysis within the Eclipse IDE and ran it on several examples from the ASP.NET framework and the Eclipse JDT framework. We show that the specifications capture common constraint paradigms such as lists of associated objects, order of operations, and associated fields of a class. We also show that the analysis detects errors that occur from a local misuse of the framework in real world examples. (Section 5)

2 Example Constraints

In this section, we will explore two examples and describe the general properties of framework constraints. We will use these examples to motivate a specification language for semantic constraints and an analysis of plugin code. The first example is from ASP.NET, a web application framework that allows developers to create web pages that work together to form a complete application. The second example is from the Eclipse Java Development Toolkit framework. This framework allows developers to parse Java code and retrieve an abstract syntax tree complete with type bindings.

2.1 ASP.NET: `DropDownList` Selection

The ASP.NET framework allows developers to create web pages with user interface controls on them. These controls can be manipulated programatically through the callbacks provided by the framework. Developers can respond to control events, add and remove controls, and change the state of controls.

One task that a developer might want to perform is to programmatically change the selection of a drop down list. The ASP.NET framework provides us with the relevant pieces, as shown in Figure 1. [1] Notice that if we want to change the selection of a `DropDownList` (or any

[1]To make this code more accessible to those unfamiliar with C#, we are using traditional getter/setter syntax rather than properties.

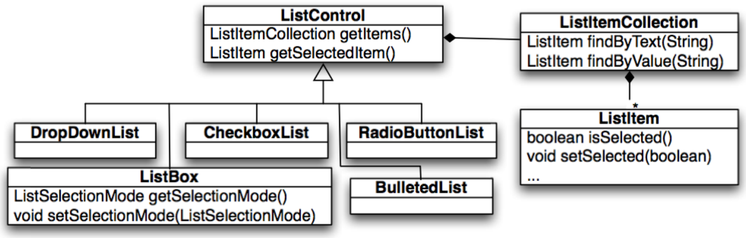
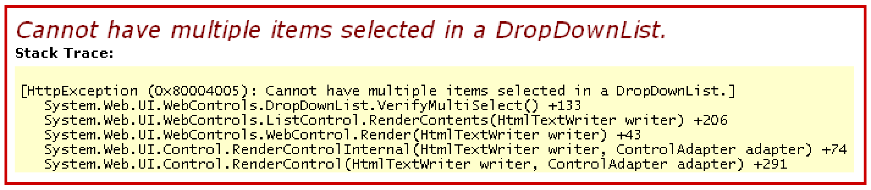

ListControl
ListItemCollection getItems()
ListItem getSelectedItem()
ListItemCollection
ListItem findByText(String)
ListItem findByValue(String)
*
ListItem
boolean isSelected()
void setSelected(boolean)
...
DropDownList
CheckboxList
RadioButtonList
ListBox
ListSelectionMode getSelectionMode()
void setSelectionMode(ListSelectionMode)
BulletedList

Figure 1: ListControl Class Diagram



Cannot have multiple items selected in a DropDownList.

Stack Trace:

```
[HttpException (0x80004005): Cannot have multiple items selected in a DropDownList.]
   System.Web.UI.WebControls.DropDownList.VerifyMultiSelect() +133
   System.Web.UI.WebControls.ListControl.RenderContents(HtmlTextWriter writer) +206
   System.Web.UI.WebControls.WebControl.Render(HtmlTextWriter writer) +43
   System.Web.UI.Control.RenderControlInternal(HtmlTextWriter writer, ControlAdapter adapter) +74
   System.Web.UI.Control.RenderControl(HtmlTextWriter writer, ControlAdapter adapter) +291
```

Figure 2: Error with partial stack trace from ASP.NET

other derived `ListControl`), we have to access the individual `ListItems` through the `ListItemCollection` and change the selection using `setSelected`. Based on this information, a developer might naïvely change the selection as shown in Listing 1. Her expectation is that the framework will see that she has selected a new item and will change the selection accordingly.

This code breaks an important framework constraint. A `DropDownList` must have exactly one item selected. If this code is run, an interesting error occurs, as shown in Figure 2. The error message clearly describes the problem; a DropDownList had more than one item selected. An experienced developer will realize that setting the selected item did not deselect the existing one, though an inexperienced developer might be confused because she did not select multiple items.

Listing 1: Incorrect selection for a DropDownList

```
DropDownList list;

private void Page_Load(object sender, EventArgs e)
{
   ListItem newSel;
   newSel = list.getItems().findByValue("foo");
   newSel.setSelected(true);
}
```

The stack trace is more interesting though; it does not point to the code where we made the selection. In fact, the entire stack trace is from framework code; there is no plugin code referenced at all! The framework called the plugin developer's code, the plugin returned control back to the framework, and then the framework discovered the error. The program control could go back and forth several times before finally reaching the check that triggered the exception. Since we don't know exactly where the problem occurred or even what object it occurred on, the developer must search his code by hand to determine where the erroneous selection occurred.

Listing 2 shows correct code for this task. We must deselect the current selection before making a new selection, and we must also perform the tasks in this order, otherwise, `getSelectedItem()` may return the newly selected item, rather than the old selected item.

A plugin developer might also make the correct sequence of calls, but on the wrong objects. It is not unusual for there to be multiple, related, `DropDownLists`. Listing 3 looks very similar to a correct program, but it makes the calls on the wrong objects.

Why didn't the `setSelected` method just assert that something was already selected? The problem is that `ListItem` does not know about its parent control, so it cannot be responsible for enforcing this constraint. Even if it had a reference to the parent, it would have to change the selection based upon the type of the control. A `DropDownList` requires exactly one item to be selected, but a `CheckboxList` allows zero or more items to be selected. Other lists, such as the `ListBox`, might even change their constraint according to how they are being used. The selection constraint is eventually checked by the derived `ListControl`, but only in a much later callback, just before the `DropDownList` is rendered on the web page.

Listing 2: Correctly selecting an item using the API

```
DropDownList list;

private void Page_Load(object sender, EventArgs e)
{
   ListItem newSel, oldSel;
   oldSel = list.getSelectedItem();
   oldSel.setSelected(false);
   newSel = list.getItems().findByValue("foo");
   newSel.setSelected(true);
}
```

Listing 3: Selecting on the wrong DropDownList

```
DropDownList listA;
DropDownList listB;

private void Page_Load(object sender, EventArgs e)
{
   ListItem newSel, oldSel;
   oldSel = listA.getSelectedItem();
   oldSel.setSelected(false);
   newSel = listB.getItems().findByValue("foo");
   newSel.setSelected(true);
}
```

There are alternate designs that avoid this problem. For example, the selection could be controlled by the `ListControl` and the mechanism could change for each derived class, or a separate class could define the selection mechanism for a `ListItemCollection`. Each of these designs have trade-offs in other places; they either limit extensibility or add complexity to the design. Short of a discussion with the designers, it is difficult to guess why they chose this particular set of trade-offs. Finally, it is possible that the designers simply overlooked this problem. This type of issue is frequently overlooked, but it is difficult to rectify the problem after the framework has become an industry standard.

2.2 Eclipse JDT: Retrieving Type Bindings

The Eclipse `ASTParser` allows a developer to retrieve an abstract syntax tree from Java source. The source may be from an `ICompilationUnit`, an `IClassFile`, or even a `char[]` that represents valid Java source. In any case, the developer simply creates an `ASTParser`, calls `ASTParser.setSource` with their source object, and retrieves the root node of the abstract syntax tree using `ASTParser.createAST`. The developer can now walk the tree and visit each of the expressions.

One of the most useful features in the Eclipse JDT framework is the ability to request type bindings for an expression. This is done by calling `Expression.resolveTypeBinding()` on the expression of interest. The resulting object captures the complete typing information for that expression, including the ability to look-up the original type definition and check for substitutability against other types.

By default, the parser does not resolve the type bindings, as this is an expensive operation. A developer must specifically request the type bindings to be resolved by calling `ASTParser.-resolveTypeBindings(boolean)`. If this is not done, then all type bindings returned from `Expression.resolveTypeBinding()` will be null.

There are two interesting details resolving the bindings:

1. After a call to `createAST`, the type binding flag is reset back to false. If the `ASTParser` is reused again, then the developer must recall the method `resolveTypeBindings` if they want type bindings. Presumably, this is to prevent unnecessary memory usage unless the developer explicitly requests it again.

2. If the source object is a `char[]`, then the type bindings will not be resolved at all unless the `ASTParser` is also provided with a compilation unit name and a project by calling `setUnitName(String)` and `setProject(IJavaProject)` respectively. Without this information, the parser does not know what classpath to use to look up the types. If the plugin does not provide this information, then the `ASTParser` will simply not resolve the bindings and any calls to `Expression.resolveTypeBinding()` will return null.

The second issue is particularly tricky, and there have been several queries about it on the `news.-eclipse.tools.jdt` newsgroup. Many developers are expecting to use the parser as shown in Listing 4. It has not occur to these developers that they need a classpath as they are using only primitive types. The developers then receive a `NullPointerException` on line 9 when they try to use the type bindings, but they are unsure why they are null in the first place.

The framework could have thrown an exception from within `createAST`, but instead it tried to perform the functionality to the best of its ability. As type bindings could be null for other reasons, it appears that the framework developers presumed that the plugin developers should be

Listing 4: Incorrect usage of `ASTParser`

```
public String getBindingName {
   ASTParser parser = ASTParser.newParser(AST.JLS3);
   Expression exp;

   parser.setKind(ASTParser.K_EXPRESSION);
   parser.setResolveBindings(true);
   parser.setSource("5 + 9 / 7".toCharArray());
   exp = (Expression)parser.createAST(null);
   return exp.resolveTypeBinding().getQualifiedName();
}
```

checking for null anyway. However, if the plugin is dependent on the type bindings and provides otherwise valid source, it doesn't expect this behavior.

While this behavior is documented, developers don't know what documentation to look at. They get the null result from `Expression.resolveTypeBinding()`, but the documentation about this is written in the `ASTParser` class. Several developers had this problem and posted on the newsgroup about it. There are also several newsgroup posts where a developer asked about another problem, and then a framework developer helpfully pointed out that their code would also not work because of this issue.

2.3 Properties of Framework Constraints

In both of these examples, the plugin developers broke a framework constraint unknown to them. They had used the framework in a way which seemed intuitive because they had a slightly misshapen view of what the framework was doing. Additionally, the cause of the problem was not clear from the runtime error. In the first example, the message was helpful, but the stack trace did not come from the plugin code. In the second example, both the stack trace and the error message were unhelpful.

Based on these examples and several others mined from the ASP.NET developer forum and the Eclipse newsgroup, we have identified three interesting properties of framework constraints. These properties are specific to framework constraints and imply a set of challenges that a solution must overcome. Previous work, as discussed in Section 6, does not cover all of these properties.

Framework constraints involve multiple classes and objects. Unlike library and protocol constraints, which typically involve only one object, framework constraints frequently span across several objects. These objects may not even know about other objects which they share a constraint with.

Framework constraints are non-local. In both of these examples, the method that triggers the error is defined in a class that has no knowledge of the framework constraint.

Framework constraints have semantic properties. Framework constraints are not only about structural concerns such as method naming conventions or types. In each of these examples, the developer had to be aware of the order of operations and the *relationships* that objects had with each other. In the `DropDownList` example, the developer had to perform the selection operations in a particular order. Additionally, the items had to both be children of the same `DropDownList`.

3 Developer-defined Relations over Objects

When a developer programs to a framework, the primary task is not about creating new objects or data. In many cases, programming in this environment is about *manipulating the relationships between existing objects*. Every time the plugin receives a callback from the framework, it is implicitly notified of the current relationships between objects. As the plugin calls framework methods, the framework changes these relationships, and the plugin learns more about the state of these relationships. Every method call, field access, or branch test gives the plugin more information. Even when the plugin needs to create a new object, it is frequently done by calling abstract factory methods that create and set up object relationships for the plugin.

Both of the examples displayed this mechanism of interaction. In the ASP.NET example, all the objects are created by the framework, and the plugin simply changes their properties. The Eclipse example created only one object, the `ASTParser`, and all others were created by the framework.

In this section, we introduce three constructs based on the interactions between frameworks and plugins. These constructs allow framework developers to specify relations that occur at runtime and the framework constraints that depend on them. A static analysis can then read the plugin and use the specifications to find locations where the plugin breaks a framework constraint. The specifications and the static analysis meet our initial goals of finding a modular, low cost solution which provides directed errors. We will use the `DropDownList` selection example to motivate this section, though the specifications and analysis have also been completed for the other example.

The first construct, *relation annotations*, specifies how framework operations change associations between objects. When a plugin calls a framework method, the analysis knows to change the relation according to the annotations defined on the method. By tracking this information, the analysis will be able to build up a context for every expression in the plugin.

The second construct, *constraints*, represent a precondition for a framework operation. Before using the operation it constrains, the plugin must be able to meet a set of requirements on the relations. A constraint can operate on non-local methods, and it might only be triggered under certain conditions.

The third construct, *relation logic statements*, specifies implicit knowledge about how relations interact with each other. This knowledge is used to show that a constraint can be met, even if some relations were not explicitly declared. This allows developers to specify constraints across classes which are not directly involved in the constraint, for example, defining that all items in a particular List or Map are taking part in a relation.

A `.rels` file in each package provides a place to specify the types of the relations, the constraints, and the relation logic statements. The syntax for this file and the constraints and relation logic is listed in Figure 3. We will refer back to this figure as necessary in this section. The relation annotations are written directly onto the framework source code.

$$
\begin{array}{rcl}
relsFile & ::= & \overline{relationDef}\ \overline{logicStatement}\ \overline{constraint} \\
relationDef & ::= & name(\overline{\tau}) \\
logicStatement & ::= & P\ \texttt{allows}\ \overline{relPred} \\
constraint & ::= & name\ op\ \texttt{with}\ Q_{trigger}\ \texttt{requires}\ Q_{req}\ \texttt{apply}\ \overline{relPred} \\
Q & ::= & P \mid * \\
P & ::= & P \wedge P \mid P \vee P \mid P \implies P \mid instancePred \mid equalsPred \mid relPred \\
instancePred & ::= & x\ \texttt{instanceof}\ \tau \mid !instancePred \\
equalsPred & ::= & x == x \mid x \neq x \\
relPred & ::= & name(\overline{z}) \mid !name(\overline{z}) \mid ?name(\overline{z}, x_{isTrue}) \\
op & ::= & invocation \mid constructor \mid ... \\
invocation & ::= & \tau.methodName(\overline{\tau\ x}) \\
constructor & ::= & \texttt{new}\ \tau(\overline{\tau\ x}) \\
z & ::= & \_ \mid x
\end{array}
$$

where τ is a type and x is a variable

Figure 3: Syntax of the .rels file

3.1 Relation Annotations

Relation annotations specify changes to the relations that occur after calling a framework method. The framework developer annotates the framework methods with information about how the calling object, parameters, and return value are related (or not related) after the method call. A relation annotation allows us to add or remove tuples from the calling context's relations. For example, the annotation `@Item(item, list)` creates an "Item" tuple between `item` and `list`, while `@!Item(item, list)` removes this tuple. [2] Relation annotations may refer to the parameters, primitive values, the receiver object, and the return value of a method. Additionally, parameters can be wild-carded, so `@!Item(_, list)` removes all the "Item" tuples between `list` and any other object.

A set annotation combines the previous two annotations so that the addition or removal is defined by a parameter. This allows a branch-sensitive analysis to do tests based upon the expected runtime value of a parameter. In a set annotation, the last argument determines whether the tuple will be added or removed. For example, we might annotate the method `List.contains(Object obj)` with `@?Item(obj, this, return)` to signify that this tuple is added in the true branch of a condition and removed on the false branch.

Relations are entirely user-defined and have no predefined semantics. Any hierarchy or ownership present, such as "Child" or "Item" relations, is only inserted by the framework developer. Relations are defined in the `.rels` file as a unique name with a set of types, as shown by *relationDef* in Figure 3. Currently, the analysis only supports relations with a unique name

[2] While this is not correct Java syntax, we will use it in our examples so that they map easily to the predicates as defined in Figure 3. The correct Java syntax for the add annotation appears as `@AddRel(name=``Item'', params={``item'', ``list''})`. This is the syntax used in the implementation.

and arity, though future support will include the ability to overload a relation across several types.

As an example, Listing 5 displays the relation annotations for the `DropDownList` API. Notice that multiple relation annotations can be placed on a method to signify changes to several relations, or even multiple changes to a single relation.

Section 4 describes the static analysis that tracks relations through the plugin code using the relation annotations. After calling a method, we add or remove a set of tuples from the relation, according to the relation annotations defined on that method. In this way, relations are temporal and different tuples exist in different contexts. By tracking relations through the code with this analysis, each expression in the plugin is associated with a relation context that describes the current state of the framework objects.

3.2 Constraints

Once we can track relations, we can describe the framework constraints and use *relationship predicates* to describe semantic aspects of the constraint. As noted earlier, framework constraints have both syntactic and semantic parts. We will write constraints as semantic preconditions to a framework operation, where the operation must also exist in a particular context.

As shown in Figure 3, the syntax for a constraint is:

$$name\ op\ \texttt{with}\ Q_{trigger}\ \texttt{requires}\ Q_{req}\ \texttt{apply}\ \overline{relPred}$$

where Q is a logical predicate that allows conjunction, disjunction, and implication of predicates. This should be read as "when we find an operation op that exists in a context where $Q_{trigger}$ is true, then we must also show that Q_{req} is true and we must change the relations to match $\overline{relPred}$" The predicates used in Q can be about the type of variables, equality of variables, or the tuples that are in a relation.

Like the relation annotations, predicates about relations can have three forms. The syntax $name(\overline{z})$ is used to denote that the tuple should exist in the relation, while the syntax $!name(\overline{z})$ is used to denote that the tuple should not exist. The syntax $?name(\overline{z}, x_{isTrue})$ is similar to the set annotation; it is either expected to be true or false according to the value of x_{isTrue}.

Listing 6 shows the `.rels` file for the DropDownList example. In this file there are two constraints on the operation `ListItem.setSelected(boolean select).` One is triggered when `select` is true, while the other is triggered when select is false. Additionally, they are only triggered when the `ListItem` is a child of a `DropDownList.` They use the relation `NoSelection` to ensure that they are working on the same `DropDownList` object. The final line is a relation logic specification; this construct is discussed in the next section.

3.3 Relation Logic Statements

In some cases, the relations between objects are implicit. Consider the `ListItemCollection` from the `DropDownList` example. We would like to state that items in this list are in a `Child` relation with the `ListControl` parent, but it would not make sense to annotate the `ListItem-Collection` class with this information since the class does not know about `ListControls`.

Listing 5: Partial `ListControl` API

```
public class ListItem {
  @?Selected(this, return)
   public boolean isSelected();

  @?Selected(this, select)
   public void setSelected(boolean select);

   @Text(return, this)
   public String getText();

   @Text(text, this)
  @!Text(_, this)
   public void setText(String text);
}

public class ListItemCollection
  @!Item(item, this)
   public void remove(ListItem item);

   @Item(item, this)
   public void add(ListItem item);

  @?Item(item, this, return)
   public boolean contains(ListItem item);

   @Item(return, this)
   @Text(text, return)
   public ListItem findByText(String text);

  @!Item(_, this)
   public void clear();
}

public class ListControl {
   @Items(return, this)
   public ListItemCollection getItems();

   @Child(return, this)
   @Selected(return)
   public ListItem getSelectedItem();
}
```

Listing 6: ListControl API relations file

```
Item(ListItem, ListItemCollection)
Items(ListItemCollection, ListControl)
Child(ListItem, ListControl)
Text(String, ListItem)
Selected(ListItem)
NoSelection(ListControl)

ListItem.setSelected(boolean select)
   with select == true ∧ Child(this, ctrl) ∧
      ctrl instanceof DropDownList
   requires NoSelection(ctrl)
   apply !NoSelection(ctrl)

ListItem.setSelected(boolean select)
   with select == false ∧ Child(this, ctrl) ∧
      ctrl instanceof DropDownList
   requires Selected(this)
   apply NoSelection(ctrl)

Items(list, ctrl) ∧ Item(item, list) allows Child(item, ctrl)
```

Relation logic statements allows us to describe these implicit relations as information we can assume at any time. This construct takes the form

$$P \texttt{ allows } \overline{relPred}$$

where P is a logical predicate that must evaluate to true for the relation predicates in $\overline{relPred}$ to be assumed as true. At any time, we can assume the existence of relation predicates defined in $\overline{relPred}$ if P is true and the relations produced are more precise than the ones currently in the context. As an example, we can write $Items(list, ctrl) \wedge Item(item, list)$ `allows` $Child(item, ctrl)$ to specify that if an item is a member of a list used by a `ListControl`, then it is a child of that `ListControl`. In the next section, we will see that relation logic is very useful for showing that triggers and requirements of constraints are true.

4 The Relation Analysis

We have designed and implemented a static analysis to track relations through plugin code and check that all operations are valid given the specified constraints. The relation analysis is a branch-sensitive, forward dataflow analysis.[3] It is designed to work on a three address code transformation

[3]By branch-sensitive, we mean that the true and false branches may receive different lattice information depending upon the condition. The transfer function on the condition is called twice, once assuming that the result is false, and

of Java source. We assume that the analysis runs in an analysis framework which takes care of the branch-sensitivity tracking and the three address code transformation. In this section, we will present the lattice and the transfer functions, along with an intuition behind the analysis and a discussion of trade-offs. Section 5 provides information about the implementation and the experience with our examples.

The relation analysis is dependent on several other analyses, including an alias analysis and a boolean constant propagation analysis. The relation analysis makes two assumptions about the alias analysis. First, it assumes that given any variable, the alias analysis can return a finite list of locations that the variable might point to. If there is an infinite list due to an assignment within a loop, this is represented as a single location, though it is marked as a "summary" node. Second, it assumes that the alias analysis can return a finite set of *all* the locations that are substitutable for a given type. It will do this automatically if we give it a "wildcard" variable, as discussed in the previous section.

4.1 Soundness and Completeness Trade-offs

We have decided to make this analysis *unsound*, that is, it will not find all the constraint violations. This is an explicit tradeoff to the benefit of expressability of constraints and concise specifications. As we will see in the design, there are two primary causes of unsoundness: aliasing and subtyping. As aliasing is entirely taken care of by the external alias analysis, this is solved by substituting in a sound alias analysis. However, sound alias analyses that currently exist would result in a combination of more false positives, more user effort, or a less modular analysis. Therefore, we have chosen to use an unsound alias analysis for the time being. As we will argue in a later section, treating subtyping in a sound way would either severely limit the expressability of the constraints, or cause an excessive number of false positives. We will address the soundness concern later in the section.

The analysis is also not complete; it can produce warnings where no error exists. Our preliminary investigations show that the false positives typically come from code which is difficult to read; refactoring the code into a cleaner state appears to remove the false positives. We will be investigating the extent of false positives in real codebases in later work, along with how to minimize them further.

4.2 The Relationship Lattice

We define a *relationship* as a single tuple in a relation among a set of references. If the relation `Item` is defined as

$$Item = \{\texttt{ListItem} \times \texttt{ListItemCollection}\}$$

and we have the following references in scope

$$\{item1 : \texttt{ListItem},\ item2 : \texttt{ListItem},\ list : \texttt{ListItemCollection}\}$$

once assuming that it is true. This is not a path-sensitive analysis, the condition result is not saved for use after the branches merge together.

then the possible relationships are

$$\{Item(item1,\ list),\ Item(item2,\ list)\}$$

At any point, every possible relationship maps to a four-point lattice with the values `true`, `false`, `unknown` (the most conservative, top element of the lattice) and the (optimistic) bottom element of the lattice. The default state for a relationship is `unknown`, and the bottom value is never used.

The relation analysis uses the lattice σ, defined as a tuple lattice mapping relationships to the boolean lattice above.

$$\sigma = \overline{\{name(\overline{\ell}) \mapsto value\}}$$

Notice that as more references enter the context, there are more possible relationships, and the height of σ grows. Even so, the height is always finite as there is a finite number of references ℓ and a finite number of relations. As the flow functions are monotonic, the analysis always reaches a fix-point.

4.3 Transfer Functions

The transfer functions are responsible for two tasks; they must check that the operation is valid, and they must apply any specified relationship changes to the lattice. We will present only the transfer function for the operation $x_{ret} = x_{this}.m(\overline{x})$ since the algorithm to perform these two tasks is more interesting than the transfer functions themselves.

Before running the analysis, we translate all relation annotations into the same form as constraints. A set of relation annotations can be thought of as a constraint with no trigger and no requirement. Therefore, the method annotated

```
public class ListItem {
   @!Text(_, this)
   @Text(text, this)
   public void setText(String text);
}
```

becomes the constraint

```
ListItem.setText(String text)
    with * requires * apply Text(text, this), !Text(_, this)
```

The analysis presumes that all constraints, including translated annotations, are in the set `Constraints`, while the set `RelLogics` contains all the relation logic statements defined.

The transfer function for the operation $x_{ret} = x_{this}.m(\overline{x})$ is quite simple. The function `constraintCheck` returns a list of changes to make to the lattice σ. The transfer function applies these changes and returns the new lattice.

$f_{x_{ret}=x_{this}.m(\overline{x})}(\sigma) =$
    let $\overline{r \mapsto v} = constraintCheck(\sigma, x_{ret} = x_{this}.m(\overline{x}))$

```
    in $\sigma[\overline{r \mapsto v}]$
```

The interesting work occurs in `constraintCheck`, shown in pseudocode in Listing 7, with helper functions defined in Listings 8 and 9. Before describing this algorithm in detail, it is helpful to see the logical intuition behind it. As we stated before, the algorithm must check that an operation is valid and retrieve the relationship changes. To check whether an operation is valid, the following predicate must be true for each constraint under the lattice σ

```
$\lambda\ op.\ \lambda\ opVarToLocBindings.$
 $\forall\ trigVarToLocBindings.$
   $op$ matches $op_{cons} \wedge [opVarToLocBindings][trigVarToLocBindings]Q_{trigger}$
   $\implies \exists\ reqVarToLocBindings.$
     $[opVarToLocBindings][trigVarToLocBindings][reqVarToLocBindings]Q_{req}$
```

That is, given an operation and the variable bindings used in it, show that for all possible variable bindings where $Q_{trigger}$ is true, there must exist some set of bindings where Q_{req} is true. In this intuition, we assume that the maps `XXXVarToBindings` are only over the free variables for `XXX`. For example, if a variable in Q_{req} is also used in $Q_{trigger}$ or op_{cons}, we use the existing binding rather than creating a fresh variable.

4.4 Technical Details

The `constraintCheck` function (Listing 7) runs over every specified constraint or translated annotation. It first checks whether the invoked method matches the declared operation based on the static types of x_{this} and $\overline{x}$. If so, it asks the alias analysis for all possible configurations for the references to the variables. For each configuration, it checks if the trigger predicate is true under the current lattice σ, the available relation logic, and the bindings γ. If the trigger is true, the algorithm calls `getRels` to produce the relationships with the appropriate bindings and lattice values. (Listing 8) Additionally, the algorithm checks whether Q_{req} is true by calling `hasRequired` and produces an error if it is not. (Listing 9)

Throughout the pseudocode, we will use γ to represent a mapping from variables x to references ℓ. The function `getLabelMaps`, not shown, produces this mapping. The function takes a set of variables that we would like to find references for. For each variable, it asks the alias analysis for all possible locations that the variable could point to. If the variable is in the scope of the operation, it takes this into account and returns locations based upon the bound variable of the operation. It then takes the cross product of all these lists; therefore producing a set of maps from variables to references. This represents all the possible *configurations* of the variables initially passed in. We can later use a configuration to bind variables to references when we instantiate a relationship.

Listing 8 displays the pseudocode for the two functions which generate relationships. The function `getRels` takes a list of relationship predicates, a configuration γ, and a bool that states whether the alias analysis found only one possible configuration. Lines 4-7 finds every "add" relationship predicate in $\overline{relPred}$. It determines that the final value of all relationships created from each predicate will be `true` only if there was one configuration and `unknown` otherwise.

Listing 7: Main constraint analysis algorithm

```
$\overline{r \mapsto v}$ constraintCheck($\sigma$, $x_{ret} = x_{this}.m(\overline{x})$) {
    changes := $\phi$
    for ($op_{cons}$ with $Q_{trigger}$ requires $Q_{req}$ apply $\overline{relPred} \in$ Constraints) {
        if ($x_{ret} = x_{this}.m(\overline{x})$ matches $op_{cons}$) {
            $\overline{y}$ := $\overline{x} \cup \{x_{ret}, x_{this}\} \cup$
                (freeVars($Q_{trigger}$) − freeVars($op_{cons}$))
            $\overline{\gamma}$ := getLabelMaps($\overline{y}$)
            $\overline{\gamma}$ := [freeVars($op_{cons}$) / ($\overline{x} \cup \{x_{ret}, x_{this}\}$)]$\overline{\gamma}$
            for ($\gamma \in \overline{\gamma}$) {
                if ($\sigma; RelLogics; \gamma \vdash Q_{trigger}$) {
                    changes := changes $\cup$ getRels($\overline{relPred}$, $\gamma$, $||\overline{\gamma}|| = 1$)
                    if (!hasRequired($\sigma$, $\gamma$, $Q_{req}$)) then
                        Error at $op$, could not match $Q_{req}$
                }
            }
        }
    }
    return changes
}
```

It calls `instantiateRel` to instantiate the relationships from the predicate with the provided context and map them to the provided value. Lines 9-12 do a similar operation for all the "remove" relationship predicates, and lines 14-19 handle the "set" relationship predicates, including setting their value to a specified variable.

The main operation of `instantiateRel` is lines 27-29, where it does the substitution on the relationship predicate to produce a relationship and map it to a value. If the value is a variable, line 28 calls out to a boolean constant propagation analysis to determine if it has a concrete value. Lines 24-26 simply take care of any wildcard variables in the relationship predicate. Any variable which was not bound by this point is a wildcard, and the alias analysis will return all references it knows about that have the same type.

Listing 9 checks whether the predicate Q is true under the lattice σ, configuration γ, and any available relation logic statements. The function returns `true` if it can find *any* bindings that make Q true. If it is not possible to make Q true with γ, then the function returns `false`.

In both `checkConstraints` and `hasRequired`, we reference the judgement

$$\overline{\sigma; P \texttt{ allows } \overline{relPred}}; \gamma \vdash Q$$

While we do not show the rules for this judgement, they work approximately as expected. If Q does not hold in σ, then the set of relation logic statements is used to attempt to produce relationships that make Q true. There are two interesting points about this judgement:

1. The use of a relation logic does not change the original lattice σ. Relation logic statements are used implicitly, but they never cause a change. This allows the relationships produced

Listing 8: Generate relation elements given a configuration

```
$\overline{rel \mapsto value}$ getRels($\overline{relPred}$, $\gamma$, isSingleConfiguration) {
    rels := $\phi$

    **for** ($name(\overline{y}) \in \overline{relPred}$) {
        value := isSingleConfiguration ? **true** : unknown
        rels := rels $\cup$ instantiateRel($name(\overline{y})$, $\gamma$, value)
    }

    **for** ($!name(\overline{y}) \in \overline{relPred}$) {
        value := isSingleConfiguration ? **false** : unknown
        rels := rels $\cup$ instantiateRel($name(\overline{y})$, $\gamma$, value)
    }

    **for** ($?name(\overline{y}, z) \in \overline{relPred}$) {
        value := isSingleConfiguration ? $z$ : unknown
        rels := rels $\cup$ instantiateRel($name(\overline{y})$, $\gamma$, value)
    }
    **return** rels
}

$\overline{rel \mapsto value}$ instantiateRel($name(\overline{y})$, $\gamma$, value) {
    rels := $\phi$

    wildCards := $\{y : y \in \overline{y} \wedge y \notin dom(\gamma)\}$
    configs := $\gamma\times$ getLabelMaps(wildCards)}
    **for** ($\overline{x \mapsto l} \in$ configs) {
        **if** (value = z)
            value := getBooleanValue([$\overline{x \mapsto l}$]z)
        rels := rels $\cup([\overline{x \mapsto l}]name(\overline{y}) \mapsto$ value)
    }
    **return** rels
}
```

Listing 9: Check whether any configurations can make Q true

```
bool hasRequired($\sigma$, $\gamma_{start}$, $Q$) {
    $\overline{\gamma_{local}}$ := getLabelMaps(freeVars($Q$) − dom($\gamma_{start}$))
    for ($\gamma_{local} \in \overline{\gamma_{local}}$) {
        if ($\sigma; RelLogics; \gamma_{start}, \gamma_{local} \vdash Q$)
            return true;
    }
    return false
}
```

from a relation logic statement to go away automatically if the generating predicate, P, is no longer true.

2. Any relationship produced from a relation logic statement must be *strictly more precise* than the relationship's value in σ. This means that relationships can move from unknown to true, but they can not move from false to true. This property prevents the logical inference engine from thrashing on relation logic statements of the form

$$name(x, y) \texttt{ allows } !name(x, y)$$

4.5 Soundness Revisited

We identified two major sources of unsoundness in this system. The first is the alias analysis, which could be substituted for a sound alias analysis in the future, as described earlier. The second source of unsoundness is subtyping. As an example of this, consider a plugin which incorrectly selects an item from a DropDownList, except the list is of the supertype, ListControl. Since the object is of type ListControl, the trigger clause of the first constraint in Listing 6 will not be true, and the constraint will never trigger an error. While there are ways to make this sound, the costs are so large that we have chosen to allow unsoundness into the analysis. We will discuss each of these options in turn.

We could require that specifications for a class be consistent with the specifications of the supertype. We would have to write the constraint assuming the type is ListControl, and then override this constraint when we know the type is DropDownList. For this to work, we would have to strengthen the constraints for ListControl so that they can catch the issues currently detected by the DropDownList constraints. However, ListControl simply does not have these constraints on it, so we would not be able to express this issue.

Another option is to proactively assume that a ListControl object may be any possible subtype of ListControl, so we should check constraints as though it is any subtype. However, this would produce many false positives when the ListControl is actually a derived class that has a different constraint, such as CheckboxList. Additionally, this could allow conflicting constraints between the subtypes.

This leads to the observation that DropDownList is not following the principle of behavioral subtyping. It has added preconditions to methods that the base class did not require. Therefore, a DropDownList is not always substitutable where a ListControl is used! While frustrating, this appears to be a common problem with frameworks. Inheritance was used here rather than composition because the type is structurally the same, and it is almost behaviorally the same. In fact, the methods on DropDownList itself do appear to be behaviorally the same. However, the subtype added a few constraints to *other* classes, like the ListItem class.

While another framework design might have followed behavioral subtyping, there are trade-offs involved with regard to reusability and extensibility. Frameworks frequently use inheritance for the reusability and extensibility attributes, but they do not assume that the substitutability is pure. It is not uncommon for a subtype to have more constraints that its supertype or to have completely conflicting constraints. For this reason, we cannot achieve soundness without limiting the expressability of the specifications or producing excessive false positives. Based upon our preliminary examples, we believe this will not be a major concern in practice.

5 Implementation and Experience

We implemented the analysis in the Crystal dataflow analysis framework, an Eclipse plugin developed at Carnegie Mellon University for statically analyzing Java source. Crystal provides capabilities for analyzing source in three address code form, running a branch-sensitive analysis, reading specifications from annotations, and specifying dependencies between analyses. For the implementation of this analysis, we also used a boolean constant propagation analysis and a naïve alias analysis. In future revisions, either of these could be replaced with better analyses in order to improve the results; the relation analysis is only dependent on the interfaces to these analyses.

We ran the analysis on several examples from the Eclipse JDT framework, as described earlier in Section 2.2. After misusing the framework ourselves, we discovered that this issue appeared several times on the Eclipse JDT newsgroup. The examples are based on the newsgroup postings and our own experience.

Listing 10: Partial Eclipse JDT relations file

```
Bindings(ASTParser)
Project(IJavaProject, ASTParser)
UnitName(IJavaProject, ASTParser)
SourceStr(String, ASTParser)
SourceComp(ICompilationUnit, ASTParser)
SourceClass(IClassFile, ASTParser)

ASTParser.createAST(IProgressMonitor monitor)
    with Bindings(this) ∧ SourceStr(_, this)
    requires Project(_, this) ∧ UnitName(_, this)
    apply !Bindings(this)
```

The relations file for the Eclipse JDT is in Listing 10, and the annotated API is in Listing 11. The analysis worked correctly in most of the examples. The one drawback we discovered was that we received false positive or false negatives when a required object was not in scope. For example, one of the Eclipse JDT constraints, not shown here, required that an `ASTNode` have a relationship with an `AST` object. The plugin, however, did not have any `AST` objects in scope at all, even though this relationship did exist globally.

Future revisions of the analysis could address this problem with two changes. First, it should be possible for the framework to declare what relationships at the point which the callback occurs. This would have provided the correct relationships in the previous example, and it should be relatively straightforward to annotate the interface of the plugin with this information. Second, an inter-procedural analysis on only the plugin could handle the case where the relationship goes out of scope for similar reasons, such as calls to a helper function.

Listing 11: Partial `ASTParser` API

```java
public class ASTParser {
    @UnitName(unitName, this)
   @!UnitName(_, this)
    public void setUnitName(String unitName);

    @Project(project, this)
   @!Project(_, this)
    public void setProject(IJavaProject project);

    @SourceComp(source, this)
   @!SourceComp(_, this)
   @!SourceClass(_, this)
   @!SourceStr(_, this)
    public void setSource(ICompilationUnit source);

    @SourceClass(source, this)
   @!SourceComp(_, this)
   @!SourceClass(_, this)
   @!SourceStr(_, this)
    public void setSource(IClassFile source);

    @SourceStr(source, this)
   @!SourceComp(_, this)
   @!SourceClass(_, this)
   @!SourceStr(_, this)
    public void setSource(char[] source);

   @?Bindings(this, resolve)
    public void setResolveBindings(boolean resolve);
}
```

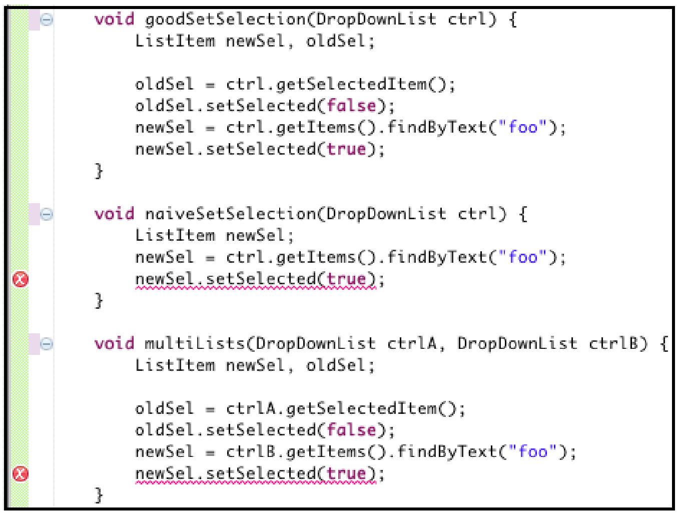
We also converted the `DropDownList` example from ASP.NET into Java. The analysis ran on methods that used the API in various ways, both correctly and incorrectly. These examples are based upon an author's past experience with ASP.NET, including defects encountered in real development. It was annotated as shown in Listing 5, and the `.rels` file is shown in Listing 6

Listing 12: Selecting on the wrong `DropDownList`

```
1 void multiLists(DropDownList ctlA, DropDownList ctlB) {
2     ListItem newSel, oldSel;
3
4     oldSel = ctlA.getSelectedItem();
5     oldSel.setSelected(false);
6     newSel = ctlB.getItems().findByText("foo");
7     newSel.setSelected(true);
8 }
```

The most interesting example is shown in Figure 12. This example is syntactically correct, but semantically incorrect. The example calls all the right methods in the right order, but it does so on two different `DropDownList` objects. The constraint analysis detects the difference because while it has the relationship $NoSelection(ctlA)$ at line 7, it does not have the required relationship $NoSelection(ctlB)$. This example would only be correct in the case where ctlA and ctlB alias one another. The current alias analysis heuristically assumes that arguments don't alias, but it could be replaced by an alias analysis which takes into account programmer annotations similar to the `restrict` keyword in C.

Figure 4 shows a screenshot of Crystal running on three examples from the `DropDownList` API. An error marker in the left margin shows that the tool detected a violation of a framework constraint.



```
void goodSetSelection(DropDownList ctrl) {
    ListItem newSel, oldSel;

    oldSel = ctrl.getSelectedItem();
    oldSel.setSelected(false);
    newSel = ctrl.getItems().findByText("foo");
    newSel.setSelected(true);
}

void naiveSetSelection(DropDownList ctrl) {
    ListItem newSel;
    newSel = ctrl.getItems().findByText("foo");
    newSel.setSelected(true);
}

void multiLists(DropDownList ctrlA, DropDownList ctrlB) {
    ListItem newSel, oldSel;

    oldSel = ctrlA.getSelectedItem();
    oldSel.setSelected(false);
    newSel = ctrlB.getItems().findByText("foo");
    newSel.setSelected(true);
}
```

Figure 4: Screenshot of the tool. Framework constraint errors are indicated in the left margin

6 Future Work

Currently the analysis points the plugin developer to the error location, but it does not provide any suggestions for how to fix the error. The developer has to read the constraint to understand the problem. We are considering adding the ability to request suggestions from the analysis. This would require the analysis to determine the point of failure in the requirement, and then determine what other operations would make that requirement true. The suggestion would be based upon the relation context right before the error and the specification of the constraint that caused the error. As an example, a possible suggestion for Listing 12 might read:

```
Suggestion: Insert before newSel.SetSelected(true)
   ListItem listItem1 = ctlB.GetSelectedItem();
   listItem1.SetSelected(false);
```

Notice that the suggestion refers to `ctlB`, which it retrieved from the relation context at the error site.

As noted in the previous section, we will also explore how to span the analysis across multiple methods. We anticipate that this will only be used locally within a plugin, and therefore marking only starting relationships at the plugin entry points will provide enough information. Future case studies across large frameworks will determine whether this feature is necessary or feasible.

7 Related Work

Some of the original work in frameworks [3] discussed using design patterns as a way of describing frameworks. Later research has looked at formalizing design patterns and extracting design patterns from code[4, 5, 6]. Patterns alone cannot completely specify a framework. While they provide information about high-level interaction mechanisms, they do not describe the temporal framework constraints shown in our examples.

SCL [7, 8] allows framework developers to create a specification for the structural constraints for using the framework. The specifications we propose focus on semantic constraints rather than structural constraints. Some of the key ideas from SCL could be used to drive the more structural focused parts of the specifications.

Object typestates [9, 10] provide a mechanism for specifying a protocol between a library and a client. Most of the typestate work focuses on the protocol for an individual object, while the work presented focuses on multiple, interacting objects in a protocol, a concept which is difficult to model in typestates.

Some typestate work has explored inter-object typestate. This work still considers each object to have an individual typestate, though it can be affected by other objects [11] or manipulated through participation in data structures [12]. The proposed specifications differ in that they view multiple heterogeneous objects as having a shared state.

Scoped Methods [13] are another mechanism for enforcing protocols. They create a specialized language construct that requires a protocol to be followed within it. Like SCL, this is structural and does not take context into account.

Tracematches have also been used to enforce protocols, and they take semantic knowledge into account. Tracematches provide a user-friendly way to specify a temporal sequence of events that involve multiple objects. They have been used to change the program execution with a dynamic analysis [14], and they have been used to check protocols with a static analysis [15]. Tracematches also allow global checking when paired with a dynamic analysis. However, a tracematch does not separate the general knowledge about the framework from the constraint itself. In cases where multiple execution traces lead to the same constraint, a tracematch would have to specify each possibility. Since the proposed solution depends on relationships, the constraint only needs to specify which relationships it requires for each program point. This separation of the constraints from the relationships allows the specifications to be more flexible to future changes.

Bierman and Wren propose a first-class language construct for implementing bi-directional relations consistently among multiple objects [16]. Our work, in contrast, assumes a standard object-oriented implementation of relations, but allows developers to specify these relations abstractly and track them statically to ensure that framework constraints hold.

Like the proposed framework language, Contracts [17] also view the relationships between objects as a key factor in specifying systems. A contract also declares the objects involved in the contract, an invariant, and a lifetime where the invariant is guaranteed to hold. Contracts allow all the power of first-order predicate logic and can express very complex invariants. Contracts differ from the proposed specifications because they do not tie directly back to the plugin code and have a higher complexity for the writer of the contract. Finally, constraints are not checked with a static analysis.

Other research projects [18, 19, 20, 21] help plugin developers by finding or encoding known good patterns for using frameworks. The proposed work differs significantly in that it does not suggest a way to complete the task, but it finds defects once a task has been started. We see the two bodies of research as complimentary.

This work also has some overlap with formal methods, particularly in describing the relationships and invariants of code [22, 23]. These formal methods verify that the specified code is correct with respect to the specification. Instead, we are checking the unspecified plugin code against the framework's specification. Other formal methods [24, 25] focus on a detailed description of the entire system. These systems also allow developers to model the invariants between objects. However, the checkers for these systems are meant to stand on their own, without any ties to executable code. The closest work in formal methods is [26], as it also allows for framework developers to define their own constraints. All of these checkers expect to verify global properties and invariants of the system. The analysis presented in this paper checks constraints that only hold true for specific contexts, and it takes into account that the relationships between objects might change over time.

The analysis itself is similar to a shape analysis, with the closest being TVLA [27]. This work allows custom shape analyses to create new predicates between objects. The custom analysis writer must provide logic that states whether the predicate is true or false for each type of expression; this logic is similar to a transfer function. It is possible to translate the constraint specifications into logic that can be used by TVLA.

8 Conclusion

Frameworks place constraints on plugins that affect the operations which a plugin may use. These constraints can be dependent upon many interacting objects, and they can be dependent on the previous operations.

We have created a lightweight and modular way to specify framework constraints and check plugins for broken constraints. The plugin developers only run a static analysis tool, and the specifications are written entirely by framework developers. The framework developer uses *relations* to define associations between framework objects at runtime. Our static analysis tracks the tuples in these relations as they are changed by calls to the framework. The framework developer can define constraints as preconditions over relations that must be true given the current relation context. Our static analysis checks each operation in the plugin code to ensure that the precondition holds given its current set of relations and its past interactions with the framework. The analysis works on several examples pulled from the ASP.NET and Eclipse frameworks, and it has been shown to cover many interaction paradigms between frameworks and plugins.

References

[1] Johnson, R.E.: Frameworks = (components + patterns). Commun. ACM **40**(10) (1997)

[2] Jaspan, C., Aldrich, J.: Checking semantic usage of frameworks. In: Proceedings of the 4th symposium on Library Centric Software Design. (2007)

[3] Johnson, R.E.: Documenting frameworks using patterns. In: OOPSLA. (1992)

[4] G. Florijn, M. Meijers, P.v.W.: Tool support for object-oriented patterns. In: ECOOP. (1997)

[5] D. Heuzeroth, S. Mandel, W.L.: Generating design pattern detectors from pattern specifications. In: ASE. (2003)

[6] Soundarajan, N., Hallstrom, J.O.: Responsibilities and rewards: Specifying design patterns. In: ICSE. (2004)

[7] Hou, D., Hoover, H.J.: Towards specifying constraints for object-oriented frameworks. In: Proceedings of the 2001 conference of the Centre for Advanced Studies on Collaborative research. (2001)

[8] Hou, D., Hoover, H.J.: Using SCL to specify and check design intent in source code. IEEE Trans. Softw. Eng. **32**(6) (2006)

[9] DeLine, R., Fahndrich, M.: Typestates for objects. In: ECOOP. (2004)

[10] Bierhoff, K., Aldrich, J.: Modular typestate checking of aliased objects. In: OOPSLA. (2007) 301-320

[11] Nanda, M.G., Grothoff, C., Chandra, S.: Deriving object typestates in the presence of inter-object references. In: OOPSLA. (2005)

[12] Lam, P., Kuncak, V., Rinard, M.: Generalized typestate checking for data structure consistency. In: Verification, Model Checking, and Abstract Interpretation. (2005)

[13] Tan, G., Ou, X., Walker, D.: Enforcing resource usage protocols via scoped methods. Appeared in the 10th International Workshops on Foundations of Object-Oriented Languages. (2003)

[14] Walker, R.J., Viggers, K.: Implementing protocols via declarative event patterns. In: FSE, (2004)

[15] Martin, M., Livshits, B., Lam, M.S.: Finding application errors and security flaws using PQL: a program query language. In: OOPSLA. (2005)

[16] Bierman, G., Wren, A.: First-class relationships in an object-oriented language. In: ECOOP. (2005)

[17] Helm, R., Holland, I.M., Gangopadhyay, D.: Contracts: specifying behavioral compositions in object-oriented systems. In: OOPSLA. (1990)

[18] Froehlich, G., Hoover, H.J., Liu, L., Sorenson, P.: Hooking into object-oriented application frameworks. In: ICSE. (1997)

[19] Riehle, D.: Framework Design: A Role Modeling Approach. PhD thesis, Zurich (2000)

[20] Mandelin, D., Xu, L., Bod, R., Kimelman, D.: Jungloid mining: helping to navigate the API jungle. In: PLDI. (2005)

[21] Fairbanks, G., Garlan, D., Scherlis, W.: Design fragments make using frameworks easier. In: OOPSLA, (2006)

[22] Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: PLDI. (2002)

[23] Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: a behavioral interface specification language for Java. SIGSOFT Softw. Eng. Notes **31**(3) (2006)

[24] Jackson, D.: Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. **11**(2) (2002) 256–290

[25] Spivey, J.: The Z Notation: A Reference Manual. Prentice Hall (1992)

[26] Andreae, C., Noble, J., Markstrum, S., Millstein, T.: A framework for implementing pluggable type systems. SIGPLAN Not. **41**(10) (2006) 57–74

[27] Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. **24**(3) (2002) 217–298